Facultad de Ingeniería y Ciencias Hídricas
FICH
UNL

Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática

FUNDAMENTOS DE PROGRAMACIÓN

Asignatura correspondiente al plan de estudios de la carrera de Ingeniería Informática

ANEXO 2
Herramientas de Depuración

Ing. Pablo Novara - 21/04/2010

Herramientas de Depuración

Herramientas a utilizar

La depuración, según las etapas de resolución de problemas planteadas al comienzo de la asignatura, es el proceso de encontrar y corregir errores (de lógica, no de sintaxis) en un programa. Existen herramientas que nos permiten observar detalladamente como evolucionan los datos a medida que se avanza el programa, y qué acciones son las que en verdad se ejecutan. Es decir, con estas herramientas podemos pausar la ejecución en un punto dado, avanzar paso a paso observando por qué lineas del código va pasando el control del programa, observar cuanto valen determinadas variables en cada paso, etc[1]. La mayoría de los IDEs actuales incluyen herramientas para la depuración. En esta guía se utilizará el IDE *Zinjal* para los ejemplos presentados, pero los conceptos generales son aplicables también en cualquier otro entorno[2].

Es importante destacar que más allá de la idea general que dicta que la depuración sirve para encontrar y corregir errores; en muchos casos la depuración será también de utilidad aún cuando el programa funcione correctamente. Por ejemplo, cuando debemos comprender como funciona un código escrito por otra persona. Con esta guía se busca fomentar el uso del depurador como una herramienta didáctica y como un complemento indispensable para el estudiante. Si bien en los primeros pasos, la falta de experiencia con el lenguaje y el compilador puede causar confusión, una vez dominadas las habilidades básicas de depuración se acortarán sensiblemente los tiempos de desarrollo y se podrá aprovechar al máximo la capacidad del IDE. Además, para proyectos de mayor tamaño (como el proyecto final de la asignatura), en muchos casos resulta imposible realizar un seguimiento mental o una prueba de escritorio debido a la complejidad del código y los múltiples caminos de ejecución.

En esta guía se desarrollarán tres ejemplos prácticos ilustrando el manejo básico de las facilidades de depuración. Cada ejemplo debería realizarse como complemento a la práctica de una unidad de la asignatura. Es decir, el primer ejemplo corresponde a la práctica de la unidad 7 (estructuras de control), el segundo a la práctica de la unidad 8 (funciones), el tercero a la practica de las unidades 9 (arreglos), y 10 (estructuras). Para evitar mayores confusiones, se recomienda que no avance con ejemplos correspondientes a unidades que aún no han sido dadas en las clases prácticas o teóricas, porque contienen códigos y conceptos que tal vez no comprenda.

1 El tipo de depuración que se presentará aquí se puede catalogar como depuración a nivel de código fuente, dado que el programador dispone del código fuente del programa a depurar y controla el proceso utilizándolo como referencia. No es el único tipo de depuración que existe, pero sí el más utilizado durante el desarrollo de un software.

2 *Zinjal*, funciona en realidad como una interfaz visual para *gdb*, el verdadero depurador.

Consideraciones Previas

En el proceso de compilación, el código fuente se traduce en código de máquina. Sabemos que teniendo sólo el ejecutable (código de máquina) es imposible recuperar el código fuente, pero aún teniendo ambos (ejecutable y fuentes) no es trivial relacionar ambos códigos y saber por ejemplo, qué dirección de memoria le asignó el sistema operativo a una variable, o qué conjunto de instrucciones de máquina se corresponden a una linea de código. Es por esto que para que la depuración desde el código fuente sea posible, el compilador debe introducir dentro del ejecutable información adicional que le permita establecer esta relación. Este ejecutable (llamado común mente versión Debug), por lo tanto, será más grande, y eventualmente más lento[3] que el ejecutable final que el programador entregará al usuario una vez finalizado el proceso de desarrollo (versión Release). Además, una vez compilado el ejecutable en versión Debug, se debe recordar que si se modifican los fuentes, se pierde la relación que existe entre éstos y el código de máquina del ejecutable, y en este caso el depurador podría mostrar información incorrecta.

Otro detalle a tener en cuenta, es que para poder controlar correctamente un programa, en muchos casos, el depurador debe encargarse de cargarlo y ejecutarlo, por lo que el IDE presentará generalmente dos formas de ejecución: la ejecución normal, y la ejecución para depuración (que será más lenta aún para el mismo ejecutable).

3 La inclusión o no de la información de depuración no es la única diferencia entre las compilaciones Debug y Release. Hay otras diferencias (como niveles de optimización) que afectan de forma más perceptible la velocidad de ejecuación del programa.

Ejemplo Nro 1: Control básico e inspección de variables

Paso 0: El programa ejemplo

Para llevar a cabo este ejemplo utilizaremos el siguiente código:

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
     unitbuf(cout);
     int cant,n;
     cout<<"Cantidad de datos: ";
     cin>>cant;
     double sum=0;
     for (int i=0;i<cant;i++) {
          cout<<"Ingrese el dato "<<i<<": ";
          cin>>n;
          sum+=n;
     }
     double prom = sum/cant;
     cout<<"Promedio: "<<prom;
     return 0;
}
```

Cree un nuevo archivo utilizando la plantilla predeterminada y copie el código fuente del recuadro.

Este código corresponde a un programa que calcula y muestra el promedio de *n* números (donde *n* es un dato ingresado también por el usuario). La variable *sum* acumula (suma) todos los enteros ingresados, la variable *cant* contiene la cantidad de enteros a ingresar, y finalmente, *prom* guardará el promedio (sum/cant). Dentro del bucle, *i* es el contador, y *n* es una variable auxiliar para leer los datos que se deben sumar en *sum*[4].

Paso 1: Detener el programa

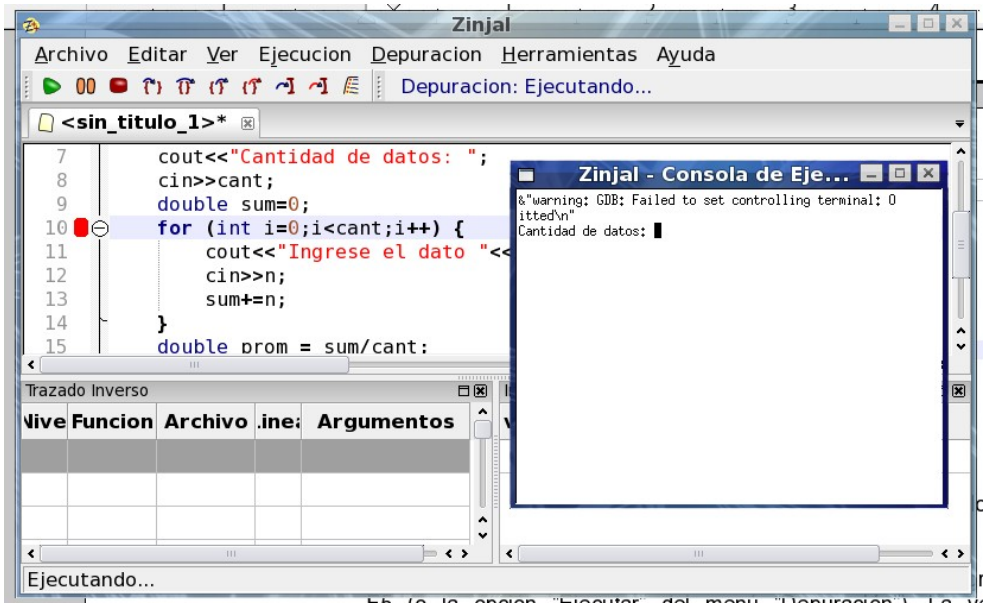
Para que podamos inspeccionar las variables, o controlar el avance del programa, primero debemos hacer que se detenga en medio de la ejecución. Para esto, antes de comenzar a ejecutarlo, debemos establecer “puntos de interrupción” (breakpoints). Estos son puntos en el código (números de linea) donde el depurador debe detener el programa. Por lo general, se indican con un circulo rojo sobre el márgen izquierdo. Para colocarlo en *ZinjaI*, puede hacer click sobre dicho márgen, o posicionar el cursor de texto en la linea de interés y presionar F8.

4 La única linea podría desconocer es la que dice “*unitbuf(cout)*;”. Esta linea se utiliza para que *cout* muestre los mensajes en consola inmediatamente, ya que el comportamiento normal es que espere a acumular varios mensajes y los muestre todos juntos (a menos que reciba antes un *endl)*. Este comportamiento podría confundir a la hora de ejecutar paso por paso el programa.

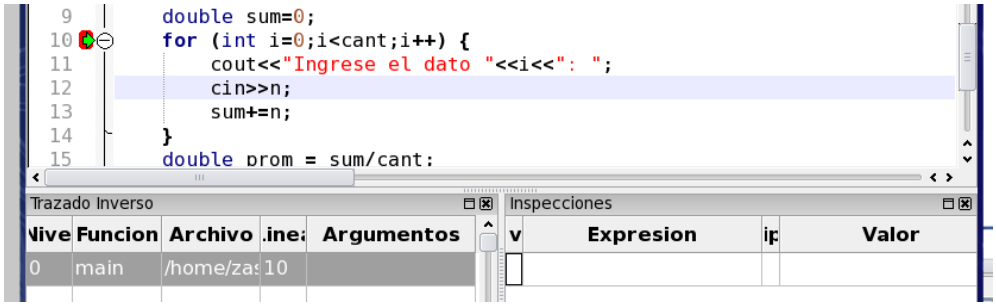
```
4  int main(int argc, char *argv[]) {
5      unitbuf(cout);
6      int cant,n;
7      cout<<"Cantidad de datos: ";
8      cin>>cant;
9      double sum=0;
10     for (int i=0;i<cant;i++) {
11         cout<<"Ingrese el dato "<<i<<": ";
12         cin>>n;
13         sum+=n;
14     }
15     double prom = sum/cant;
```

Pruebe colocar un punto de interrupción en la línea 10 (donde comienza el bucle *for*).

Una vez colocado el punto de interrupción, ejecute el programa con la tecla F5 (o la opción “Ejecutar” del menú “Depuración”). La ventana de *Zinjal* desplegará dos nuevos paneles: el panel de trazado inverso y el panel de inspecciones. Inmediatamente, el programa comenzará a correr normalmente y le solicitará que ingrese el primer dato.

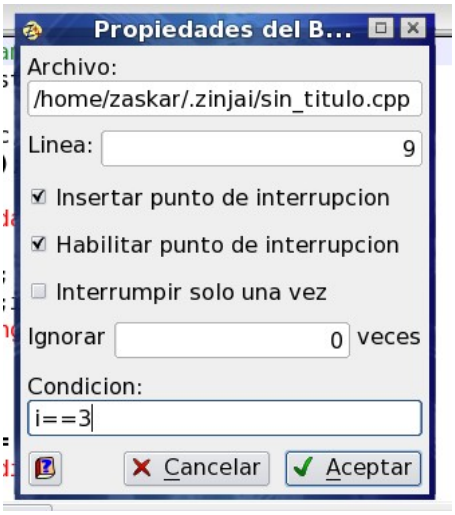


Ingrese 5 y presione *Enter*. Observará que el programa se detiene luego de leer el dato y la ventana de *Zinjal* pasa al frente (o parpadea en la barra de tareas). En el margen izquierdo del código encontrará una flecha verde sobre el punto de interrupción. Esta flecha indica dónde se ha detenido el programa. Cuando se marca una línea, quiere decir que el programa se detuvo justo antes de ejecutar esa línea.



Debe notar que no puede detener el programa en cualquier línea. Algunas, como comentarios, declaraciones de variables o lineas en blanco, no tienen correspondencia con ningún fragmento del ejecutable, por lo que no son puntos de interrupción válidos. Si define un punto de interrupción en una posición inválida, el depurador se detendrá en realidad en la siguiente posición válida.

Los puntos de interrupción, pueden tener una condición asociada, de forma que el flujo del programa, si pasa varias veces por la misma línea, sólo se detenga si se cumple dicha condición. Por ejemplo, para detenerse sólo en la cuarta iteración del bucle, la condición sería *i==3*. Para definir una condición, haga click sobre el punto de interrupción (en el margen) manteniendo presionada la tecla *Shift.*

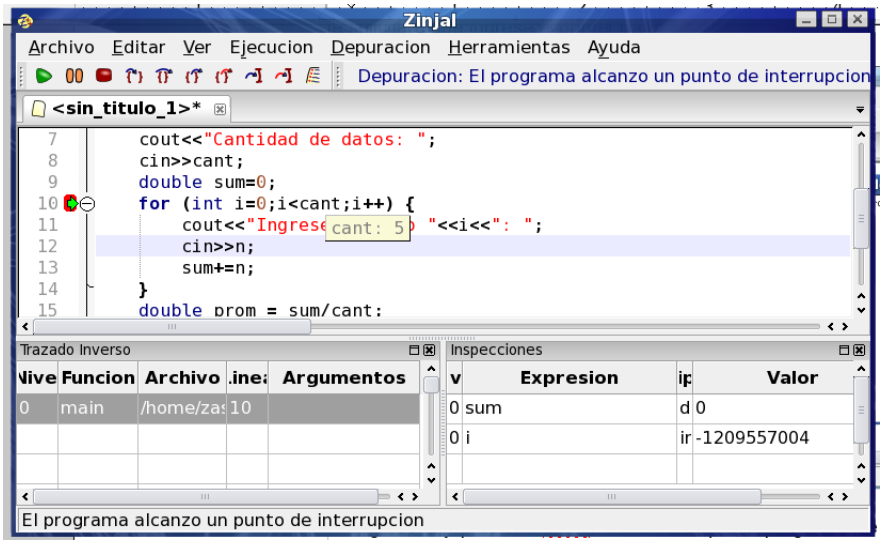


Existen otras dos formas de detener el programa sin colocar un punto de interrupción: una es generar una señal (hay un botón de pausa en la barra de herramientas de *Zinjal* para *GNU/Linux*, o se puede forzar presionando Ctrl+C en la consola de ejecución en *Windows*); la otra es generando un error (por ejemplo, acceso a una posición de memoria inválida).

Paso 2: Inspeccionar variables

Una vez que se ha detenido el programa, se pueden inspeccionar los contenidos de las variable. Existen dos formas básicas de hacerlo:

- Colocar el puntero del ratón sobre el nombre de la variable (o seleccionar una expresión) y esperar unos segundos (en la figura se muestra el valor de *cant*).
- Agregar la variable o expresión en la columna “Expresión” del “Panel de Inspecciones” (abajo a la izquierda, en la figura se muestran las variables *sum* e *i*).



Observe que el programa se detuvo justo antes de comenzar el bucle, por lo que aún no inicializó el contador. A esto se debe el extraño valor de *i*.

El primer método es más rápido, pero el segundo muestra más información (el tipo, el ambiente, etc.), y permite modificar el valor de una variable haciendo doble click sobre el mismo. Además, las entradas en la tabla de inspecciones permanecen allí cuando avanza el programa, por lo que en la próxima detención se actualizarán mostrando los nuevos valores[5].

Paso 3: Continuar la ejecución

En este ejemplo, podemos continuar la ejecución de dos formas básicas. Una es avanzando un solo paso (una línea en el código); la otra es avanzando hasta el próximo punto de interrupción.

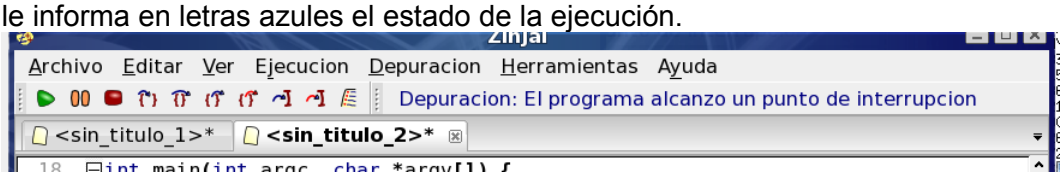
Para avanzar paso por paso, utilice la tecla F7 (*step over* en el menú "Depuración"). Cada vez que presione F7 el programa ejecuta una línea de código.

Para continuar ejecutando normalmente presione F5. Esta acción retomará la ejecución continua hasta que el programa alcance otro punto de interrupción. Si el programa no alcanzase nunca otro punto de interrupción se ejecutará hasta finalizar. En este caso, la consola de ejecución se cerrará automáticamente sin esperar a que presione una tecla.

Pruebe continuar la ejecución paso por paso y observe como varían las variables (recuerde que al llegar al paso que contiene el *cin* deberá volver a la consola de ejecución para ingresar un valor).

Para detener definitivamente el programa sin finalizar la ejecución presione Shift+F5 (o haga click en "Detener" en el menú "Depuración").

Recuerde que en todo momento, la parte derecha de la barra de herramientas le informa en letras azules el estado de la ejecución.



5 Se actualizan cuando el programa se pausa, nunca mientras este se ejecuta.

Ejemplo Nro 2: Funciones y ámbitos

Paso 0: El programa ejemplo

Para llevar a cabo este ejemplo utilizaremos el siguiente código:

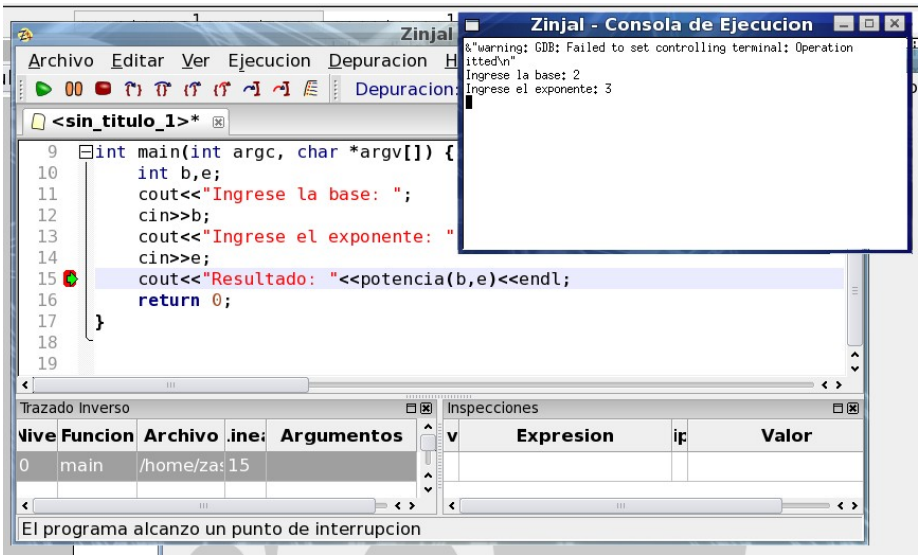
```
#include <iostream>
using namespace std;

int potencia(int base, int expo) {
     if (expo==0) return 1;
     else return base*potencia(base,expo-1);
}

int main(int argc, char *argv[]) {
     int b,e;
     cout<<"Ingrese la base: ";
     cin>>b;
     cout<<"Ingrese el exponente: ";
     cin>>e;
     cout<<"Resultado: "<<potencia(b,e)<<endl;
     return 0;
}
```

Cree un nuevo archivo utilizando la plantilla predeterminada y copie el código fuente del recuadro.

Este código corresponde a un programa que calcula una potencia entera de un número entero. Para ello utiliza la forma recursiva $a^n = a \cdot a^{n-1}$.

Paso 1: Entender el trazado inverso

Coloque un punto de interrupción en la línea 15 (donde está la llamada "*potencia(b,e);*"), y presione F5 para ejecutar el programa hasta este punto. Cuando solicite base y exponente ingrese 2 y 3 respectivamente.

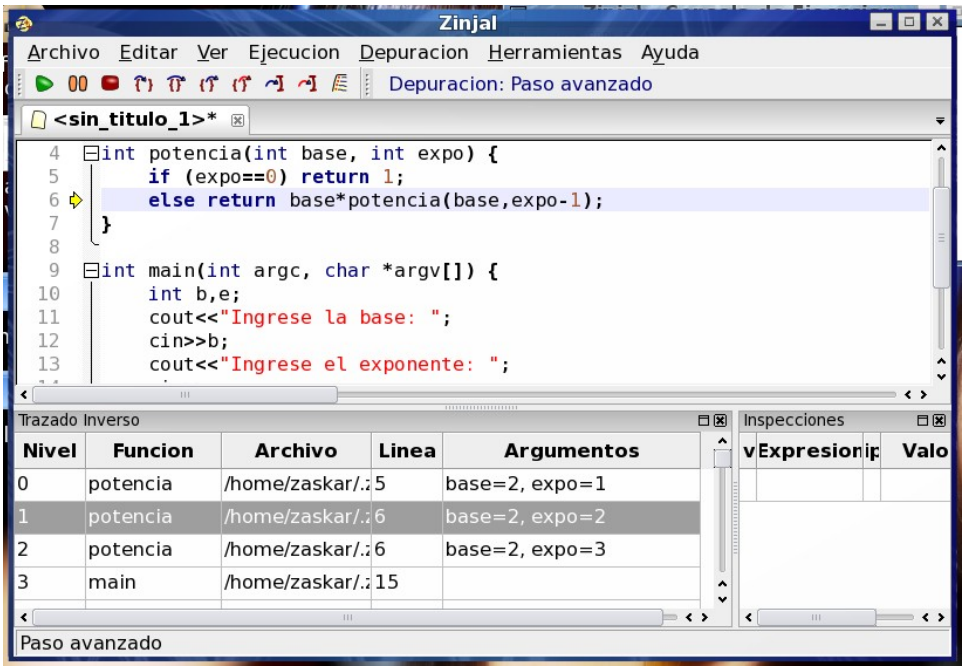


La ejecución debería detenerse justo antes de calcular el resultado. La linea que corresponde ejecutar a continuación contiene la llamada a la función potencia, por lo que en realidad, la próxima linea de código a ejecutar debería ser la primer línea de la función potencia (primero hay que evaluarla para luego poder mostrar el resultado). En estos casos, tenemos dos formas de continuar la ejecución: *step over* y *step in*.

La primera (*step over*, con la tecla F7) es la que utilizamos hasta ahora, avanza a la siguiente linea dentro del ámbito actual. En este caso, avanzará a la siguiente linea del *main*, realizando la evaluación de la función y mostrando el resultado en un solo paso.

La segunda forma de avanzar (*step in*, con F6), nos permite meternos dentro de la función. Es decir, si presionamos F6 el depurador avanzará hasta la primer línea de la función “potencia”, y nos permitirá analizar qué ocurre dentro de la misma.

Presione F6 y observe el contenido del panel de trazado inverso (abajo a la derecha, *backtrace* en Inglés). En este panel tenemos en la primer linea la función actual y el punto donde realmente estamos detenidos en la ejecución, y en las otras lineas podemos observar cómo se llegó a este punto, y qué otras funciones estan “esperando” a que se resuelva esta llamada para poder continuar. En este caso, la primer línea indicará la función *potencia* y la segunda la función *main*. Puede hacer doble click sobre cada función para dirigirse a ese punto del código (observe que la flecha sobre el margen que indica la posición será verde cuando esté en el nivel más interno, y amarilla cuando se seleccione otro). Además, la tabla muestra una columna con los argumentos de la función. Presione F6 cuatro veces y observe como se “apilan” llamadas a *potencia*, con diferentes argumentos (recursividad).

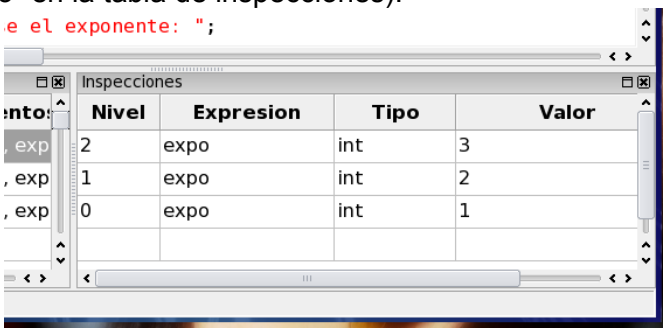


Paso 2: Identificar el ámbito de las inspecciones

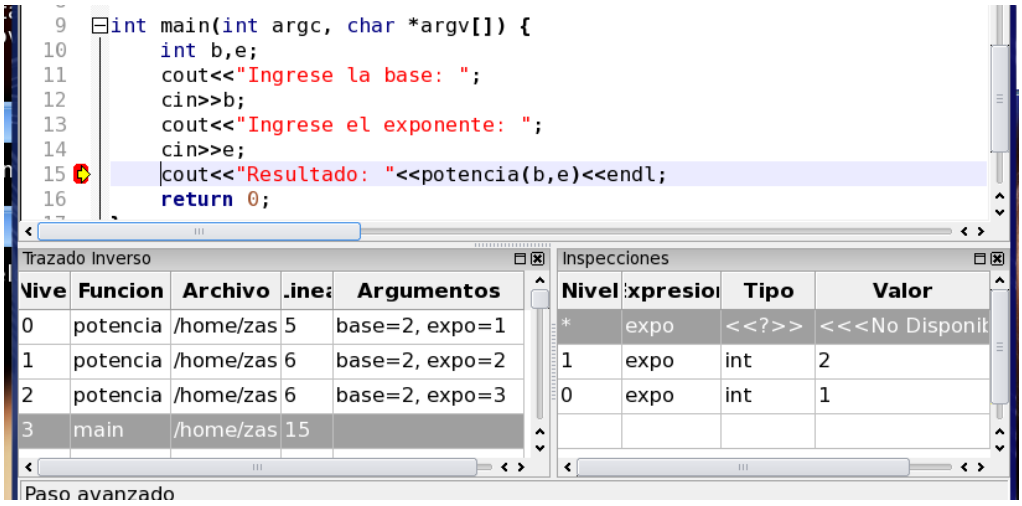
Ahora aparece un detalle más a tener en cuenta al usar la tabla de inspecciones. Se pueden encontrar distintas variables con el mismo nombre si

estas están en distintos ámbitos (en este caso cada función define un ámbito o *scope*). Cuando se ingresa una inspección en la tabla, ésta se asocia al ámbito que se encuentre seleccionado (el que marca la flecha verde/amarilla del margen). Se puede observar en la tabla de inspecciones una columna llamada "nivel" que indica en qué nivel (en relación a la tabla de trazado inverso) está el ámbito de esa inspección.

Pruebe inspeccionar la variable *expo* en distintos ámbitos (seleccione el ámbito con doble click en la tabla de trazado inverso, y luego ingrese la expresión "expo" en la tabla de inspecciones).



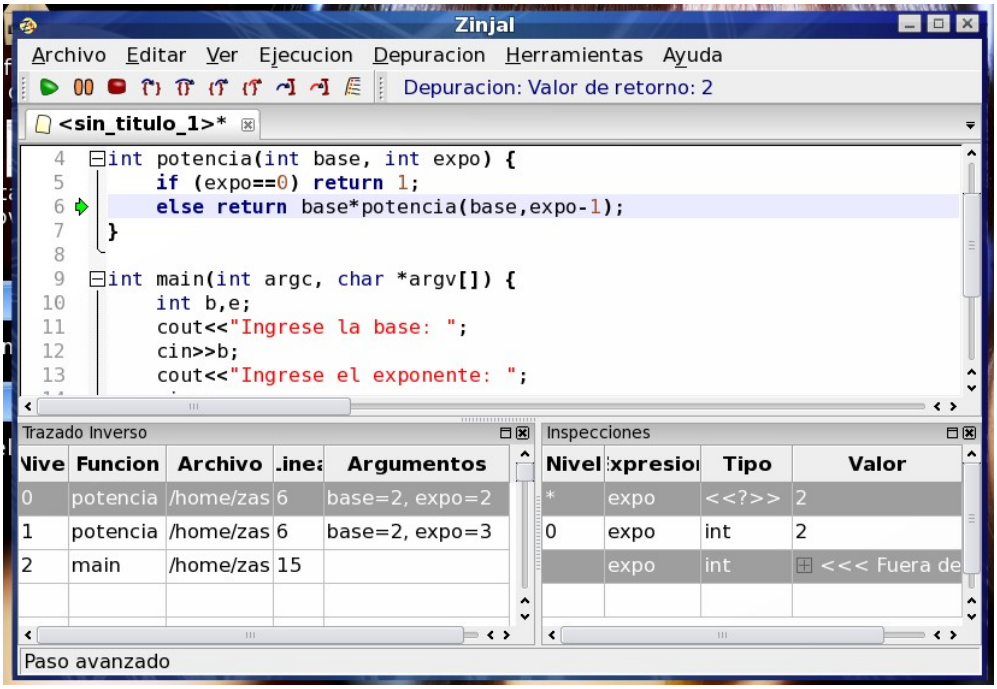
En *Zinjal* existe una segunda forma de ver las inspecciones (muchos depuradores solo cuentan con esta segunda forma), en la cual las inspecciones no tienen un ámbito asociado, sino que se evalúan en el ámbito seleccionado, y al cambiar la selección, cambian su valor. Para estas expresiones, la columna "Nivel" contiene un asterisco (*). Para convertir una expresión de un tipo en otra (es decir, asociar o desasociar con un ámbito), puede hacer doble click en la columna "Nivel". Pruebe convertir una de las expresiones, y seleccionar diferentes funciones en el trazado inverso (incluyendo main, donde la variable no existe).



Paso 3: Continuar la ejecución

Finalmente, para continuar la ejecución, tenemos ahora nuevas alternativas. Una particularmente útil es indicarle al depurador que debe continuar ejecutando hasta salir del ámbito actual, es decir, hasta finalizar la función (siempre considerando la función del nivel 0). Para esto debe utilizar *step out* (Shift+F6). Al utilizar esta acción, en la barra de estado de la depuración (el texto en azul a la derecha de la barra de herramientas) encontrará el valor de retorno que arrojó

la función al finalizar. Pruebe este método seleccionando el nivel 0 en el trazado inverso y presionando Shift+F6. Observe que en el trazado inverso hay ahora un nivel menos, y en la parte superior izquierda de la pantalla se le indica que la función finalizó retornando el entero 2.



El cursor de ejecución queda posicionado en la llamada a la función que acaba de finalizar.

Ejemplo Nro 3: Visualización de estructuras de datos

Paso 0: El programa ejemplo

Para llevar a cabo este ejemplo utilizaremos el siguiente código:

```
#include <iostream>
#include <iomanip>
using namespace std;

struct registro {
     int n1,n2;
     double d;
};

void mostrar(registro *arreglo, int n) {
     for (int i=0;i<n;i++)
          cout<<i<<right
              <<setw(5)<<arreglo[i].n1<<"  "
              <<setw(4)<<arreglo[i].n2<<"  "
              <<arreglo[i].d<<endl;
}

int main(int argc, char *argv[]) {
     registro lista[10];
     for (int i=0;i<10;i++) {
          lista[i].n1=rand()%1000;
          lista[i].n2=rand()%1000;
          lista[i].d=double(rand()%1000)/1000;
     }
     mostrar(lista,10);
     return 0;
}
```

Cree un nuevo archivo utilizando la plantilla predeterminada y copie el código fuente del recuadro.

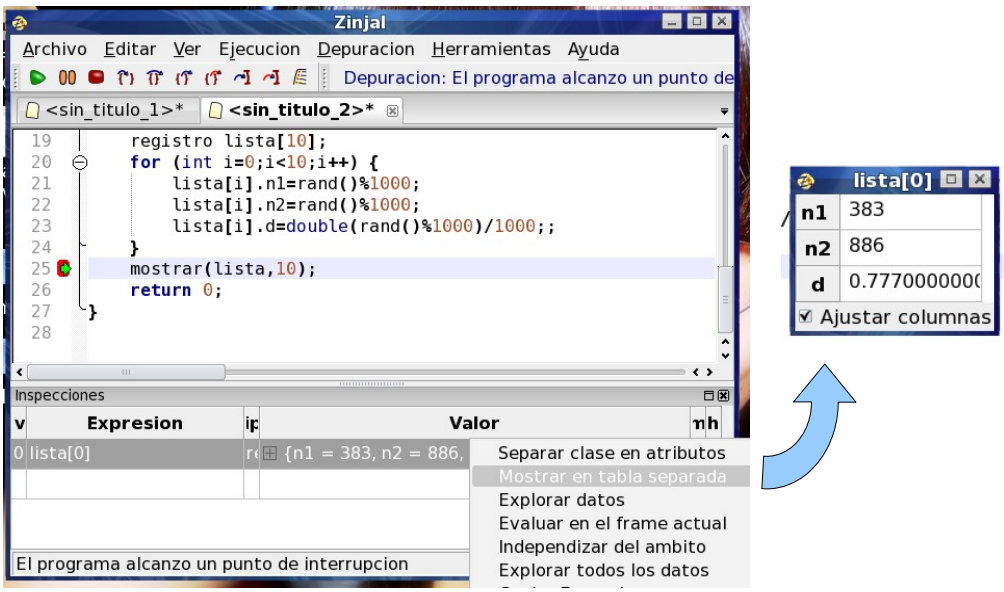
El programa declara una estructura *registro* con dos campos (un entero *n* y un real *d*), una función *mostrar* que imprime en pantalla un arreglo de registros y una función *main* que genera un arreglo estático de 10 elementos aleatorios, y luego lo muestra utilizando la función *mostrar*.

Paso 1: Inspección de estructuras

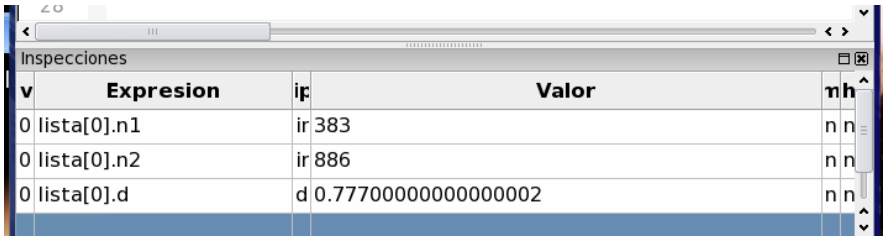
Coloque un punto de interrupción en la linea 25 (la llamada a la función *mostrar*) y presione F5 para ejecutar hasta ese punto. Allí tendremos cargados diez registros aleatorios. Ingrese la inspección “lista[0]” en la tabla de inspecciones para observar el contenido del primer registro.

El valor de la inspección contiene los tres campos separados por comas. Hay dos formas de observarlas con mayor comodidad:

1) Haga click sobre la misma con el botón derecho del ratón y seleccione “Mostrar en tabla separada” en el menú contextual. Se desplegará una nueva ventana con la estructura desplegada en una tabla.



2) Haga doble click sobre el valor de la inspección en la tabla de inspecciones. Esta acción separa una inspección compuesta en múltiples inspecciones más simples (un struct en sus atributos); y puede realizarse siempre que la inspección tiene un ámbito asociado, y muestra el símbolo + precediendo su valor.



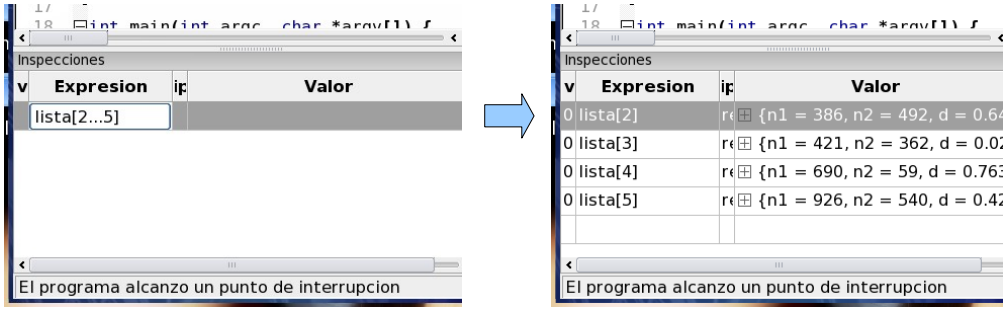
Paso 2: Inspección de arreglos

Borre las inspecciones de la tabla (seleccionándolas y presionando la tecla delete/suprimir, o con la opción “Limpiar tabla de inspecciones” del menú contextual.

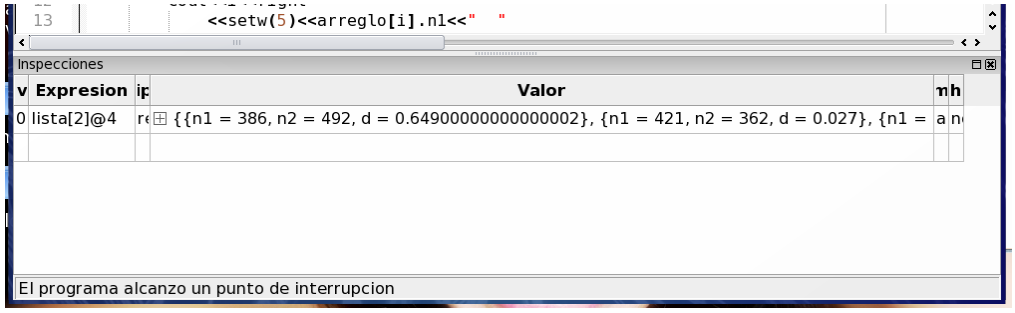
Ingrese ahora la inspección “lista” y observe como se muestra toda la lista en un solo renglón. De forma similar a las estructura registro del paso 1, puede hacer doble click para “desarmar” la inspección en diez inspecciones (una por elemento), o seleccionar la opción “Mostrar en tabla separada” del menú contextual para visualizarla en una nueva ventana.

Además de estas dos opciones, existen dos alternativas más para visualizar solo una parte del arreglo, por ejemplo desde los índices 2 a 5 (cuatro elementos), ingresando una notación especial. Para ello puede introducir alguna de las siguientes expresiones:

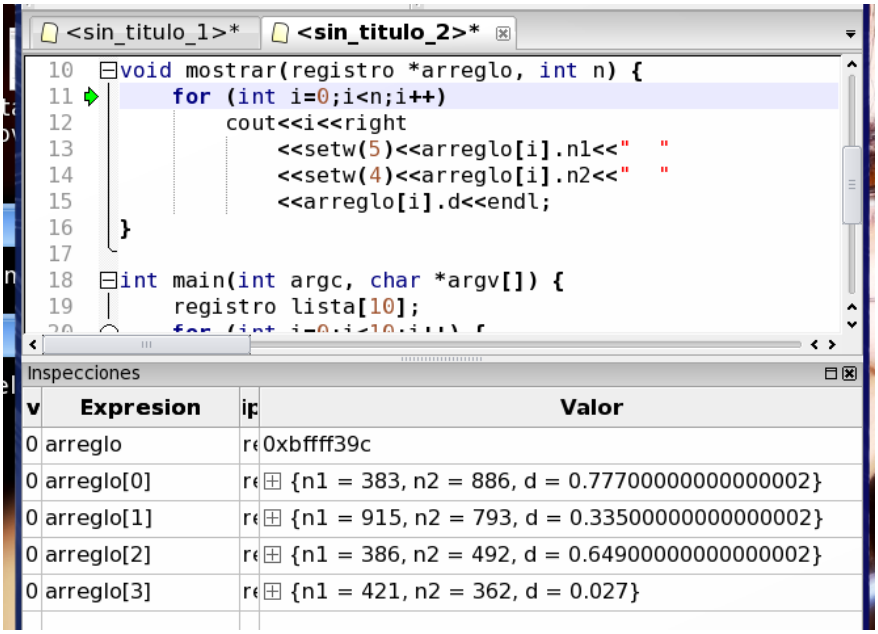
1) “lista[2...5]”: el uso de tres puntos suspensivos generará automáticamente 4 inspecciones variando el índice entre 2 y 5.



2) “lista[2]@4”: el uso de la arroba significa que se deben mostrar 4 variables ubicadas en la memoria en forma contigua (como ocurre con los arreglos lineales). A diferencia del primer método, de esta forma se genera una sola inspección con la lista de cuatro elementos.



Finalmente, presione F6 para ingresar en la función *mostrar* y una vez dentro ingrese la inspección “arreglo”. Observe que en este caso no vemos los elementos, sino una dirección de memoria (la del primer elemento, donde comienza el arreglo). Esto se debe a que dentro de la función, es imposible saber el tamaño del arreglo a partir del mismo (por eso necesitamos el parámetro *cant*). Sin embargo, aún puede mostrar sus elementos ya sea como inspecciones o en tabla separada con cualquiera de los métodos descriptos anteriormente.



Resumen de comandos

Se resumen a continuación los comandos presentados para controlar la ejecución[6]:

- Colocar/quitar punto de interrupción (F8)
- Detalles del punto de interrupción (Ctrl+F8)
- Iniciar depuración o continuar ejecución (F5)
- Avanzar a la siguiente línea, *step over* (F7)
- Ingresar en la función, *step in* (F6)
- Salir de la función, *step out* (Shift+F6)
- Ejecutar hasta donde se encuentra el cursor (Shift+F7)

Existen otras posibilidades no desarrolladas en esta guía, principalmente destinadas a alterar la ejecución (como salir de una función sin ejecutar lo que falta, continuar ejecutando desde un punto arbitrario, deshacer la ejecución (solo en *GNU/Linux*), etc.). Una vez familiarizado con el depurador, puede investigar por su cuenta estas funcionalidades.

Conclusiones

Ya habrá observado que las herramientas de depuración pueden ser de gran ayuda a la hora de encontrar errores. Sin embargo, estas herramientas no señalan el problema, sino que sólo se limitan a exponer ante el programador la evolución del programa. Es el programador quien debe saber donde colocar los puntos de interrupción y qué variables inspeccionar para poder encontrar el problema. Por lo general, tendrá un sospecha o intuición acerca del fragmento de código que causa un problema, y realizará pruebas con datos simples de modo que pueda comparar sus respuestas calculadas con lápiz y papel con los valores observados en las inspecciones en cada paso. Poder encontrar el lugar y las variables adecuadas, aunque no es trivial al principio, es una “habilidad” que se mejora con la práctica, y que requiere un buen conocimiento del lenguaje y del problema por parte del programador.

6 Los iconos podrían cambiar si no utiliza el tema predeterminado.